

PATENT ABSTRACTS OF JAPAN

(11)Publication number : 2000-066904

(43)Date of publication of application : 03.03.2000

(51)Int.Cl.

G06F 9/46

(21)Application number : 10-236031

(71)Applicant : CANON INC

(22)Date of filing : 21.08.1998

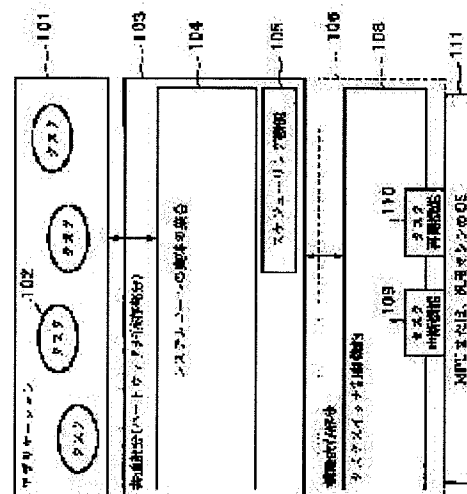
(72)Inventor : HIRAHARA ATSUSHI
BESSHO MASATAKA
IWABUCHI YOICHI
YAMADA JUNJI

(54) METHOD FOR CONTROLLING MULTITASK AND STORAGE MEDIUM

(57)Abstract:

PROBLEM TO BE SOLVED: To enable application to execute the same operation by having the same interface specification on all platforms in multitask control for controlling an application program including plural tasks.

SOLUTION: A common part 103 executes scheduling related to the execution of plural application tasks 102 included in application 101 and issues a task control instruction such as task interruption processing and task restarting processing to a machine kind depending part 106. The depending part 106 controls a task based on the task control instruction issued from the common part 103. In this case, when the operation environment of the application 101 is on an MPU in a real machine, a task interruption function 109 and a restart function 110 directly rewrite the contents of a task control block. On the other hand, when the operation environment is on a general OS, the task is controlled by using a system call supported by the general OS.



(19)日本国特許庁 (J P)

(12) 公 開 特 許 公 報 (A)

(11)特許出願公開番号
特開2000-66904
(P2000-66904A)

(43)公開日 平成12年3月3日(2000.3.3)

(51)Int.Cl.
G 0 6 F 9/46

識別記号
3 4 0

F I
G 0 6 F 9/46

テマコード*(参考)
3 4 0 B 5 B 0 9 8

審査請求 未請求 請求項の数11 O L (全 17 頁)

(21)出願番号 特願平10-236031

(22)出願日 平成10年8月21日(1998.8.21)

(71)出願人 000001007

キヤノン株式会社

東京都大田区下丸子3丁目30番2号

(72)発明者 平原 厚志

東京都大田区下丸子3丁目30番2号 キヤ
ノン株式会社内

(72)発明者 別所 正隆

東京都大田区下丸子3丁目30番2号 キヤ
ノン株式会社内

(74)代理人 100076428

弁理士 大塚 康徳 (外2名)

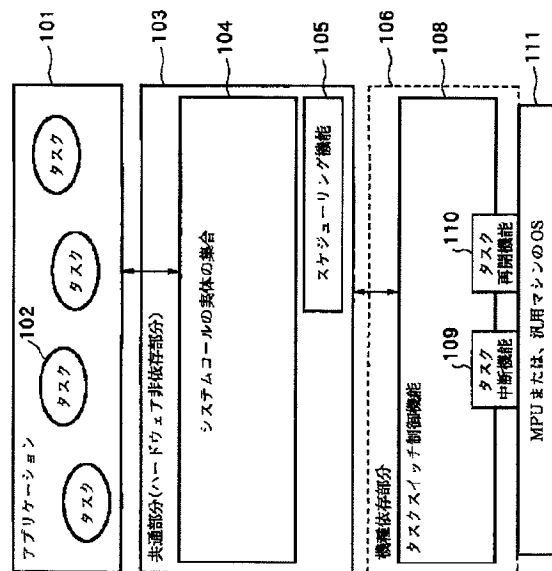
最終頁に続く

(54)【発明の名称】 マルチタスク制御方法及び記憶媒体

(57)【要約】

【課題】複数のタスクを含むアプリケーションプログラムを制御するためのマルチタスク制御において、あらゆるプラットフォーム上で同一のインターフェース仕様を持ち、アプリケーションが同一の動作を行うことを可能とする。

【解決手段】共通部分103は、アプリケーション101に含まれる複数のアプリケーションタスク102の実行に関わるスケジューリングを行い、タスク中断処理やタスク再開処理等のタスク制御指示を機種依存部分106に発行する。機種依存部分106は、共通部分103より発行されたタスク制御指示に基づいてタスクの制御を行う。ここで、当該アプリケーションの動作環境が実機のMPU上であれば、タスク中断機能109、再開機能110はタスクコントロールブロックの内容を直接書き換える。一方、動作環境が汎用OS上であれば、当該汎用OSがサポートするシステムコールを用いてタスクの制御を行う。



【特許請求の範囲】

【請求項1】 複数のタスクを含むアプリケーションプログラムを制御するためのマルチタスク制御方法であって、前記複数のタスクの実行に関わるスケジューリングを行い、タスク制御指示を発行するスケジューリング工程と、前記スケジューリング工程で発行されたタスク制御指示に基づいて、当該アプリケーションプログラムの動作環境に応じてタスクの実行を制御する実行工程とを備えることを特徴とするマルチタスク制御方法。

【請求項2】 前記タスク制御指示はタスクの中断指示及び再開指示を含み、前記実行工程は、前記スケジューリング工程で発行された中断指示及び再開指示に基づいて、当該アプリケーションプログラムの動作環境に応じてタスクの中断処理及び再開処理を実行することを特徴とする請求項1に記載のマルチタスク制御方法。

【請求項3】 前記実行工程は、前記タスク制御指示に基づいてタスクを制御するためのタスクコントロールブロックのデータを書き換えることによりタスクを制御することを特徴とする請求項1に記載のマルチタスク制御方法。

【請求項4】 前記実行工程は、前記タスク制御指示に基づいて汎用OSのシステムコールを発行することによりタスクを制御することを特徴とする請求項1に記載のマルチタスク制御方法。

【請求項5】 前記スケジューリング工程におけるタスク制御指示がタスクの生成指示及び削除指示を含むことを特徴とする請求項1に記載のマルチタスク制御方法。

【請求項6】 前記実行工程は、更に、前記アプリケーションプログラムの動作環境に応じた割込み処理が可能であることを特徴とする請求項1に記載のマルチタスク制御方法。

【請求項7】 前記実行工程は、汎用OSからの割込み信号に応じて割込み処理を実行するためのスレッドを生成し、割込み処理を実行することを特徴とする請求項6に記載のマルチタスク制御方法。

【請求項8】 前記スケジューリング工程で発行したタスク制御指示について履歴情報を保持する保持工程と、前記保持工程で保持された履歴情報を前記アプリケーションプログラムの動作環境に応じた形態で出力する出力工程とを更に備えることを特徴とする請求項1に記載のマルチタスク制御方法。

【請求項9】 コンピュータに、複数のタスクを含むアプリケーションプログラムを制御させるためのマルチタスク制御プログラムを格納する記憶媒体であって、該マルチタスク制御プログラムが、前記複数のタスクの実行に関わるスケジューリングを行い、タスク制御指示を発行するスケジューリング工程の

コードと、

前記スケジューリング工程で発行されたタスク制御指示に基づいて、当該アプリケーションプログラムの動作環境に応じてタスクの実行を制御する実行工程のコードとを備えることを特徴とする記憶媒体。

【請求項10】 前記マルチタスク制御プログラムが、前記スケジューリング工程で発行したタスク制御指示について履歴情報を保持する保持工程のコードと、前記保持工程で保持された履歴情報を前記アプリケーションプログラムの動作環境に応じた形態で出力する出力工程のコードとを更に備えることを特徴とする請求項9に記載の記憶媒体。

【請求項11】 前記スケジューリング工程のコードがC言語プログラムのテキストで記述されていることを特徴とする請求項9に記載の記憶媒体。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】本発明は、マルチタスク処理が可能なマルチタスク制御方法及び記憶媒体に関する。

【0002】

【従来の技術】一般に、リアルタイムOS（以下、RTOSという）では、実機上で動作するRTOSをワークステーションやPCなどの汎用開発マシン上で模擬実行するためのRTOSシミュレーション開発環境が存在する。

【0003】これらのRTOSシミュレーション環境には、（1）RTOSのシステムコールのスタブを用意し、アプリケーション・タスク毎の単体テストを開発マシン上で行うもの、（2）ターゲットのMPUの動作を模擬するシミュレータを用いて、ターゲット用のクロスコンパイルで生成したRTOSとアプリケーションを動作させるもの、（3）ターゲット用RTOSのハードウェア依存部分を改変して、それを汎用マシンのOS（UNIXやWindowsなど）が提供するマルチスレッド機構等を利用し、アプリケーション・タスクをスレッドとして実現するもの、などが存在する。

【0004】特に（2）、（3）の方法では、実機上のRTOSとほぼ同じ関数インターフェースを持つように作られている。これにより、実機上とほぼ同じアプリケーション・プログラムをRTOSシミュレータ上で動作させることが可能である。

【0005】また、RTOSアプリケーションのデバッグ環境として、実機上ではICEや、ROMエミュレータを介して、もしくはリモートデバッグなどの手法により、開発マシンと実機を接続し、開発マシン上で動作するソースレベルデバッガを利用するものが一般的である。また、開発マシン上で動作させるRTOSシミュレータに対するデバッグ環境としては、上記（2）の場合においては、通常、MPUのシミュレータが機械語もしくはC言語などのソースレベルのデバッガを備えてい

10

20

30

40

50

る。また、上記(1)、(3)に関しては、開発マシン上で動作させるネイティブのソースレベルデバッガを使用してデバッグ/テストを行うことができる。

【0006】ところが、RTOSアプリケーションのデバッグにおいては、これらのソースレベルデバッガを用いてブレークをかけてアプリケーションの動作を一時停止させたり、ステップ実行を行うデバッグスタイルが必ずしも効果的に行えるとは限らない。つまり、リアルタイム性が求められるこの種のアプリケーションにおいては、プログラムを停止させた瞬間にそのアプリケーションの再現性が失われてしまうからである。

【0007】そのため、近年ではアプリケーションプログラムを停止させることなく、システムコールやタスクスイッチの履歴を記録しておき、しかる後にその情報を取り出して開発マシン上などでグラフィカルに表示し、アプリケーションタスクの動きが意図したとおりであるかを解析することができるツール（以降、タスクスイッチ・ビューアと呼ぶ）が開発され普及し始めている。

【0008】また、特開平5-282160の「リアルタイム・シミュレーション開発機構」のようにダイナミック・ローディング方式を採用し、実機とシミュレーション開発環境でアプリケーションをまったく変更せずに動作させることを目的としたシステムも存在する。

【0009】

【発明が解決しようとする課題】しかしながら、上記(1)の方法では、タスク単体の振る舞いの確認を行うのみであり、タスク間の同期・通信に関わるテストを行うことはできない。また、上記(2)の方法には、実行速度が遅いために、大きな規模のアプリケーションのテストにおいてリアルタイム性がないという問題や、効率上の問題が存在する。これに対して上記(3)の方法は、シミュレーション開発環境の実現が比較的容易で、実行速度の面でも実機に最も近い振る舞いをするとされている。しかしながら、(3)の方法においても、アプリケーションの振る舞いやデバッグ環境に関して以下のような問題点が存在する。

【0010】①プラットフォーム間のアプリケーションの動作が異なる：従来のRTOSでは、実機上で動作するRTOSと開発用コンピュータの汎用OS上で動作させるRTOSシミュレータの間には、同一のアプリケーションが厳密に同一の振る舞いをすることが保証されていない。それは、シミュレーション環境では、汎用OS（UNIXやWindowsなど）が提供するマルチスレッド機構をそのまま利用してタスク（スレッド）間の同期・通信機能などを実現しているためである。すなわち、汎用OSが提供するマルチスレッド機構では、その汎用OS上でスケジューリングされるため、同時に動く他のプロセスや周辺デバイスへのアクセスの影響等で、スケジューリングの結果が変化する可能性がある。

【0011】②プラットフォーム間の移植の問題：同様

の問題で、RTOSシミュレータを動作させる汎用OSを変更する場合（例えば、UNIXからWindowsへ移植する等）、それぞれの汎用OSで提供しているマルチスレッド機構のインターフェース仕様や動作が異なるため、OSシミュレータ同士の間でもアプリケーションの動作の再現性を確保するのは困難である。また、シミュレータプログラム中に、汎用OSが提供するマルチスレッド機構のAPIが散在しているため、ポーティング（Porting：移植）が容易ではない。

【0012】③デバッグ情報（トレース情報）取得が困難：また、実機上でタスクスイッチ・ビューアなどのデバッグ装置を利用する場合は、実機上のRTOSのタスクスイッチを行う部分にトレース情報を取り出して記録する機能を追加し、記録された情報を何らかの方法（実機に用意されている物理的な通信手段：例えば、シリアル（＝シリアル通信；RS-232Cなどに代表されるビット単位の通信手法の一つ）やソケット（TCP/IPネットワークを利用するためのAPIの一つ）など）で外部の表示装置（タスクスイッチ・ビューアなど）に取り出す仕組みを用意すればよい。ところが、RTOSシミュレータにおいて汎用OSのスレッドの同期・通信機構を利用している場合、スレッドのコンテキストスイッチ（つまり、タスクスイッチ）はその汎用OS上で行われるため、タスクスイッチの履歴を収集するには汎用OSの内部にまで立ち入る必要があり、非常に困難である。従って、RTOSシミュレータのトレース情報を収集するのは容易ではない。

【0013】特開平5-282160は上記①を解決することを目的としているが、汎用OSで利用するシステムコールの種類の言及が無く、タスクスケジューリングはシミュレータを動作させる汎用OSに依存する構成となっている。また、③に関する様な、タスクスイッチのトレース情報を表示させる仕組みは持っていない。

【0014】本発明は上記の問題点を鑑みてなされたものであり、その目的は、あらゆるプラットフォーム上で同一のインターフェース仕様を持ち、アプリケーションが同一の動作を行うことを可能とするRTOSの構造を提供することにある。

【0015】また、本発明の目的は、あらゆるプラットフォーム上で同一のインターフェース仕様を持ち、アプリケーションが同一の動作を行うことを可能とするRTOS上のデバッグ環境（タスクスイッチ・ビューア等）において、あらゆるプラットフォーム上で統一的なデバッグ情報を収集することができる機構を提供することにある。

【0016】

【課題を解決するための手段】上記の目的を達成するための本発明のマルチタスク制御方法は以下の工程を備える。すなわち、複数のタスクを含むアプリケーションプログラムを制御するためのマルチタスク制御方法であっ

て、前記複数のタスクの実行に関わるスケジューリングを行い、タスク制御指示を発行するスケジューリング工程と、前記スケジューリング工程で発行されたタスク制御指示に基づいて、当該アプリケーションプログラムの動作環境に応じてタスクの実行を制御する実行工程とを備える。

【0017】また、本発明によれば、上記マルチタスク制御方法をコンピュータに実現させるための制御プログラムを格納する記憶媒体が提供される。

【0018】

【発明の実施の形態】以下、添付の図面を参照して本発明の好適な実施形態を詳細に説明する。

【0019】〔第1の実施形態〕図24は第1の実施形態におけるプログラムシミュレーション環境を提供するコンピュータ装置を示す図である。10はシミュレーション環境を提供するためのコンピュータ装置である。コンピュータ装置10は、CPU11、ROM12、RAM13、入力装置14、表示装置15、外部記憶装置16を備え、これらはバス17によって接続されている。外部記憶装置16には、実機20において実行すべきアプリケーション101と、RTOSの共通部分103と、RTOSの汎用OS依存部分106aが格納されている。

【0020】一方、実機20は制御部としてCPU21、ROM22、RAM23、I/Oインターフェース24を備え、ROM22に格納されたアプリケーション101に基づいて制御対象装置25を制御する。CPU21はROM22に格納された制御プログラム（アプリケーション101、RTOSの共通部分103、RTOSのハードウェア依存部分106b）を実行することで制御対象装置25の制御を行うことになる。

【0021】図1は第1の実施形態におけるRTOSの基本的な構成図である。101は開発対象となるアプリケーションである。102はアプリケーション・タスクであり、アプリケーション101は複数のアプリケーション・タスク102の集合である。個々のアプリケーション・タスク102はシステムコールを発行することによって、RTOSの機能を利用する。

【0022】103はハードウェアに依存しない構成要素からなるRTOSの共通部分である。104はRTOSシステムコールの実体の集合である。ここで用いるRTOSシステムコールは、ファイルシステムやネットワークインターフェースなどのハードウェア依存となるものは含まず、タスク制御やタスク間同期・通信機能（セマフォ、ロック、メールボックス、イベントフラグ、キュー、etc）など一般的なハードウェアに依存しないものを想定している。また、システムコール実行後、スケジューリング機能105によってタスク・スイッチのリスケジューリングが行われる。

【0023】106はRTOSのハードウェア/機種に

依存する構成要素群（以降、機種依存部分と呼ぶ）である。機種依存部分106において、109は現在実行中のタスクを中断する機能で、110は中断状態のタスクの実行を再開する機能である。108はタスクスイッチ制御機能で、タスク中断機能109およびタスク再開機能110を用いて、実行中のタスクを別のタスクに切り替える働きをする。なお、機種依存部分106は、図24で上述した汎用OS依存部分106a、ハードウェア依存部分106bを総称したものである。111はMPUまたは汎用OSを表す。

【0024】アプリケーション・タスク102などから、RTOSに対してシステムコールの発行が行われた場合、共通部分103ではシステムコールの実体104を実行し、その結果リスケジューリングを行う必要がある場合は、スケジューリング機能105を呼び出してリスケジューリングを行う。タスクスイッチ制御機能108はタスクの中断および再開の要求を受けてタスクの切り替えを行う。

【0025】図2は、システムコール発行からタスクスイッチまでの、共通部分による処理の手順を表したフローチャートである。システムコール関数の呼び出しを受けると、ステップS702において、タスクスイッチ制御機能108に対してタスク中断要求を発行する。そして、ステップS703においてシステム・コール本体104を実行する。本リアルタイムOSのシステムコール関数の内部は、機能的に2つの部分に別れている。それは、タスクの実行制御（タスクスイッチ：タスクの中断、リスケジューリング、タスクの再開制御を行なう）部分とシステムコール本体部分です。つまり、システムコールにはタスクの生成/破壊や、メッセージキュー、セマフォ、イベントフラグのペンド/ポストなどがあるが、それらの機能を実行する部分が「システム・コール」本体で、その機能を実行した結果に応じて、タスクの実行制御（タスクスイッチ）が行われる。そして、システム・コール本体104を実行した結果、スケジューリングの必要が生じた場合は、ステップS704からステップS705へ進み、スケジューリング機構105を呼び出してリスケジューリングを行う。なお、システムコールの種別や、状況によってはスケジューリングの必要がない場合がある。たとえば、新しいセマフォを生成するとか、現在のイベントフラグの値を調べる、といったシステムコールの実行はタスクスイッチの要因となる状態変化を起こさないので、その場合はスケジューリング機構を呼び出す必要は無い。逆に、セマフォ、イベントフラグのポスト・ペンドなどは、タスクスイッチの要因となる状態変化を引き起こすので、そのようなシステムコール実行後は必ずスケジューリングを行う。つまり、スケジューリングを必要とするシステムコールと必要としないシステムコールがあり、その判断はシステムコール本体でセットされるフラグ（スケジューリング要

10

20

30

40

50

求フラグ)により行われる。次にステップS706において、タスク制御機能108へタスク再開要求を発行し、本処理を終える。なお、次に実行を再開すべきタスクは、ステップS705のスケジューリング機構によって決定される。スケジューリングの必要が無い場合は、直前のタスクが選択される。従って、ステップS706では、現在選択されている次実行タスクを再開するための手続きを行っている。

【0026】スケジューリング機能105におけるスケジューリング・アルゴリズムには、プライオリティ順やタイムシェアリング、またはそれらを組み合わせたものなどさまざまなタイプのものが考えられる。たとえば、図3は、プライオリティ順のスケジューリングアルゴリズムを実現した一例を説明する図である。この場合、スケジューリング機能105は、実行待キュー770を保持している。実行待キュー770は、優先順位毎に実行待状態のタスクをリスト上に保持し、その並び順は、待ち行列としてFIFOのキューとして管理している。最高優先順位実行待キュー771にリンクされたタスクの中で先頭のタスク774が次実行タスクに決定される。図3のスケジューリングアルゴリズムは、優先度+待ち順となっている。実行可能なタスクは優先度毎に独立した待ち行列(FIFO)に並べられている。実行可能なタスク群の中で、もっとも優先度が高く、且つその待ち行列の先頭に位置するタスクが次実行タスクとなる。つまり、常に図3の左上のタスクが実行権を得ることになる(図3ではタスク1)。よって、図3のタスク1~タスク7における番号(数字)は実行される順番を表しているものではない。実行中のタスク(図のタスク1)が資源待ちなどの要因で実行可能タスク群から外されると、次にもっとも左上となるタスク(図ではタスク2)が実行権を得る。また、現在実行中のタスクよりも優先度の高いタスクが実行可能状態になった時は、そのタスクが実行権を得ることになる。

【0027】また、別の例として、図4は、タイムシェアリングによるスケジューリングを実現した一例を示す図である。図4によれば、スケジューリング機能105は、実行待キュー790を保持している。実行待キュー790は、待ち行列としてFIFOのキューとして管理している。実行状態にある先頭のタスク791が一定時間経過した場合、そのタスクは待ち行列キューの最後尾に送られ、キューの2番目のタスク792が先頭になりそのタスクが次実行タスクに決定される。このように、実行待ちのタスクは時間によって順次切り替えられていく。スケジューリング・アルゴリズムとしては、これらを組み合わせたものや、さらに複雑な計算式に基づいて決定する方式などが考えられる。

【0028】次に、以上のような基本構成を有するRTOSを(1)実機上と(2)UNIXワークステーション上に適用するための機種依存部分106と、スケジュー

ーリング機能105に関して説明する。なお、共通部分103は(1)、(2)とも同一のプログラム・モジュールを使用している。

【0029】(1)実機におけるRTOS

CPU21として、モトローラ社製のMC68000を搭載した実機20に対して本実施形態のRTOSを動作させるものとする。図1における機種依存部分106

(図24におけるRTOSのハードウェア依存部分106b)の構成要素について説明する。本実施形態では、メモリ(RAM23)上に、機種依存部分106がアクセスするタスク固有のコンテキスト情報(ハードウェア・レジスタ、プログラムカウンタ、タスクスタックなどを格納するタスク・コントロール・ブロック(以下、TCBと記す)を作成する。アプリケーション・タスクはRTOSの初期化ファイルの中で定義され、コンパイル時に静的に配置されているものとする。

【0030】①タスク中断機能109:図5は、実機側におけるタスク中断機能の処理の流れを表すフローチャートである。タスクスイッチ制御機能108はタスク中断要求を受けるとタスク中断機能109を実行する。タスク中断機能109は、現在実行中のタスクのコンテキスト情報を、そのタスクのTCBに保存する。具体的には、スタック上に保存されたプログラムカウンタの値を読み出し、読み出したプログラムカウンタの値、ハードウェア・レジスタの値およびスタックポインタ等をそのタスクのTCBに書き込む(ステップS502、S503)。そして、当該TCB内のタスクの状態を中断状態にセットして、中断処理を完了する(ステップS504)。

【0031】②タスク再開機能110:図6は、実機側におけるタスク再開機能109の処理の流れを表すフローチャートである。タスク再開要求を受けると、まずタスク再開機能109はタスク再開要求から再開すべきタスクのIDを獲得する(ステップS602)。そして、獲得したタスクIDに対応するタスクのTCBのタスクの状態を実行中とする(ステップS603)。更に、中断中のタスクのコンテキスト情報を、そのタスクのTCBからロードする(ステップS604)。具体的には、該当するTCBに保存されているハードウェア・レジスタ、プログラムカウンタ、スタックポインタの値を読み出し、それぞれハードウェアレジスタにセットすることによって、以前に中断した状態からそのタスクを再開することができる。

【0032】③タスクスイッチ制御機能108:実行状態のタスクを切り替える。具体的には、タスク中断要求を受けて上述のタスク中断機能109を起動させ、実行状態だったタスクのコンテキスト情報を保存して中断状態にする。また、タスク再開要求を受けた場合は、上述のタスク再開機能110を起動させ、次の実行タスクのコンテキスト情報をロードして、中断状態のタスクを実

行状態へ切り替える。

【0033】上記(1)～(3)は、機種依存部分106の構成要素であり、アセンブリ言語で書かれており、その中心となるのはタスクスイッチなどのコンテキスト操作を行う機能である。RTOSのシステムコールの実体104およびスケジューリング機能105は共通部分103に含まれる。

【0034】(2)シミュレーション環境におけるRTOS

次に、シミュレーション環境における本RTOSの実装例として、Sun Microsystems社の汎用UNIX OSである、SunOS 5.x (xはマイナーバージョンをあらわす数字。以降、略してSunOSと呼ぶ)を搭載したワークステーション上で本RTOSのシミュレータを動作させる場合を説明する。以下では、図1における機種依存部分106(シミュレーション環境側の機種依存部分であり、図24の汎用OS依存部分106aに対応する)の構成要素について説明する。なお、本実施形態では、SunOSが提供するマルチスレッドライブラリを利用して、単一プロセス内でアプリケーション・タスクをスレッドとして動作させる。

【0035】SunOSでは、スレッドの中断、再開はライブラリが提供するシステムコールによって実現される。また、アプリケーション・タスクは、メイン・スレッドの初期化ルーチンの中でシステムコールにより生成されるものとする。

【0036】①タスク中断機能109：図7はシミュレーション機側におけるタスク中断機能の処理の流れを表すフローチャートである。タスク中断要求によりタスク中断機能が実行されると、スレッドライブラリが提供するthr_suspend()システムコールにより、スレッド化されたタスクの実行を中断状態にする(ステップS1202)。タスクのコンテキスト情報はSunOSによって適切に保存される。その後、ステップS1203において、当該タスクのTCBにおける“タスクの状態”を“中断状態”にセットする。

【0037】②タスク再開機能110：図8は、シミュレーション機側のタスク再開機能の処理の流れを表すフローチャートである。タスク再開要求によりタスク再開機能が起動されると、実行を再開すべきタスクのタスクIDをタスク再開要求より獲得する(ステップS1302)。次に、実行再開すべきタスクのTCBにおける“タスクの状態”を“実行中”とする(ステップS1302)。次に、スレッドライブラリが提供するthr_resume()システムコール(ステップS1302で得たタスクIDを引数とする)により、中断状態にあるスレッド化されたタスクを実行状態にすることができる(ステップS1304)。なお、タスクのコンテキスト情報はSunOSによって適切に復帰され、指定されたタスクを以前の中断位置から実行を再開することができる。

【0038】③タスクスイッチ制御機能108：実行状態のタスクを切り替える。具体的には、タスク中断要求を受けてタスク中断機能109を起動させ、実行状態だったタスク(スレッド)を中断状態とする。また、タスク再開要求を受けてタスク再開機能110を起動させ、次に実行するタスク(スレッド)を実行状態にする。この結果、実行状態となるアプリケーション・タスクは、この系の中で常に1つとなるため(あるタスクが1つ再開すると、実行中のタスクが1つ中断するので、常に1つのアプリケーションタスクしか動作していないことになる)、SunOSによるスレッド・スケジューリングが起こっても、必ず目的とするタスク(スレッド)が実行される。このしくみにより、実機上で動く本RTOSと同一の順番でタスクが実行されることが保証される。

【0039】なお、上記①～③は、機種依存部分106の構成要素でSunOSのマルチスレッドライブラリを用いてC言語で書かれており、その中心となるのはタスクスイッチなどのコンテキスト操作を行う機能である。

【0040】以上説明したように、第1の実施形態のRTOSによれば、ハードウェアに依存する部分を必要最小限となるように切りだし、それ以外の部分は様々なプラットフォームで共通に利用できるようにC言語で書かれており、ハードウェアの違いによる影響が極力少なくなるような構造を有する。

【0041】すなわち、ハードウェアに依存する部分とは、①タスクの実行を中断する機能と、②中断されたタスクの実行を再開する機能と、これらの機能を用いて③タスクスイッチの制御をする機能であり、これらの機能を実現するプログラムモジュールは対象とするMPUや汎用OSによって書き換える必要がある。

【0042】一方、タスク制御やタスク間の同期・通信機能やタスクスイッチのスケジューリング機能等、その他すべてのRTOSの機能部分、つまり、RTOSのシステムコールを提供する階層は、すべてC言語により記述されているハードウェアに依存しない階層である。このように、タスクのコンテキスト制御等ごく限られた機能で実現する機種依存部分と、システムコール本体を実現する共通部分が完全に切り分けられた構造を持つことにより、あらゆるプラットフォーム上で同一のインターフェース仕様を持ち、アプリケーションが同一の動作を行うことが可能となる。

【0043】従って第1の実施形態によれば、RTOSの振舞に関わるシステムコールとスケジューリング機能などは、機種に依存しない共通部分として流用し、最小限に抽出されたプラットフォームに依存するプリミティブな機能のみを実装するだけで、実機とシミュレーション環境などプラットフォームが異なっても、アプリケーションに同一の振舞をさせることができる。また、機種依存部分が少なく限られた部分に集中しているので、移植作業を容易に行える。

【0044】[第2の実施形態]以下、第2の実施形態を詳細に説明する。第2の実施形態では、第1の実施形態で説明したRTOSにおいて、機種依存部分106内に動的にタスクを生成/削除する機能を追加したものである。すなわち、第1の実施形態では、タスクの動的な生成/削除を行わない(或いは、行う必要がない)システムを示したが、第2の実施形態では動的なタスクの生成/削除を行えるように拡張したシステムを示す。

【0045】(1) 実機側のRTOS

第1の実施形態と同様に、モトローラ社製のMC68000を搭載した実機に対して本RTOSを動作させるものとする。図9は、第2の実施形態による実機側のRTOSの構成を示すブロック図である。図9のRTOSは、図1で示すRTOSの基本的な構成に、タスク生成機能112bと、タスク削除機能113bを加えた構成となっている。また、114bはタスク固有のコンテキスト情報(ハードウェア・レジスタ、プログラムカウンタ、タスクスタックなど)を格納するメモリ(RAM23)上の領域(TCB)である。

【0046】①タスク生成機能112b:タスク生成機能112はタスクのコンテキスト情報を生成する。ここで、タスクのコンテキスト情報とは、タスクのある瞬間のハードウェア・レジスタ(CPU内部のレジスタ)、プログラムカウンタ、及びタスク・スタックの内容であり、タスク毎に固有の情報である。スタックポインタや、プログラムカウンタなどのコンテキストの内容は適切な値に初期化される。コンテキスト情報はあらかじめ用意されたメモリ上の空き領域に生成される(タスク情報114b)。

【0047】図10は実機側のタスク生成機能の処理を説明するフローチャートである。タスク生成要求によってタスク生成機能112が起動すると、まず、ステップS302で、新たなTCBを生成するための空きメモリ領域がRAM23に存在するかを確認する。ここで空きメモリ領域が存在しなければステップS304でエラー処理を行う。すなわち、システムコールの戻り値は、そのシステムコールの実行後の状態(成功・失敗)を示しているため、図10のステップS304の処理では「メモリ不足」を示すエラー番号を戻り値としてセットする処理を行う。一方、空きメモリ領域が存在するならば、ステップS303へ進み、空きメモリ領域から1つ分のTCBを確保する。そしてステップS304において、確保したTCBのスタックポインタや、プログラムカウンタ等を適切な値に初期化する。

【0048】②タスク削除機能113b:図11は実機側のタスク削除機能の処理を表すフローチャートである。タスク削除機能113は、タスク削除要求に応じて該当するタスクのコンテキスト情報を削除する。具体的には、そのコンテキスト情報を格納しているメモリ上の領域114を開放し、未使用状態とする(ステップS4

02)。

【0049】(2) シミュレーション環境におけるRTOS

第1の実施形態と同様、シミュレーション環境における実装例として、SunOSを搭載したワークステーション上でRTOSのシミュレータを動作させるものとする。

【0050】図12は、第2の実施形態によるシミュレーション側のRTOSの構成を示すブロック図である。図12のRTOSでは、図1で示すRTOSの基本的な構成に、タスク生成機能112aと、タスク削除機能113aが加わっている。114aはタスク固有のコンテキスト情報で、SunOSのシステム内部で管理されているメモリ上の領域である。

【0051】①タスク生成機能112a:タスク生成機能112aは、タスク生成要求に応じて、スレッドライブラリが提供するthr_create()システムコールを用いてタスクをスレッドとして生成する。スレッドのコンテキスト情報はSunOSが管理する。つまり、図12におけるタスク情報114aはSunOS内部で管理される領域である。なお、このシステムコールはスレッドを実行状態で生成するか、中断状態で生成するかをフラグにより指定できるが、本実施形態では、中断状態で生成し、タスク再開機能110によって実行を開始させるものとする。

【0052】図13はシミュレーション側のタスク生成機能の処理手順を表すフローチャートである。まず、ステップS1002において、空きメモリがあるかを判定し、空きメモリが無ければステップS1004で、図10のステップS304と同様のエラー処理を行う。空きメモリがあればステップS1003へ進み、空きメモリ領域からTCBの1つ分のメモリ領域を確保する。そして、ステップS1005にてthr_create()システムコールを呼び出して、中断状態のスレッドを生成する。その後、ステップS1006にて、当該スレッドのIDをTCBに登録する。なお、本RTOSでは、初期化時にワークスペースと呼ばれるメモリ領域を一括して確保する。そして、TCBなどのシステム管理情報は、すべてこのワークスペース内に取られる。従って実機とシミュレーション機で同じサイズのメモリ領域を(初期化時に)確保しておけば、実機/シミュレーション機間における所有メモリ領域(サイズ)は一致するので、実機と同様の空きメモリチェックを行なえる。また、RTOSの共有部分では、タスクIDでタスクの実行、中断を指示するが、タスクID、スレッドID共にTCBの中に保存されている。ここで、TCBは機種依存部分に存在するため、タスクIDとスレッドIDを1対1に対応づけることが出来る。また、図13のステップS1005ではthr_create()システムコールを用いてタスク生成を行なうが、このシステムコールによって生成されたスレ

ッドのIDが当該関数から返される。従って、このスレッドIDをタスクIDに対応付けることになる。

【0053】②タスク削除機能113a：スレッドライブラリが提供するthr_delete()システムコールにより、スレッド化されたタスクを削除する。図14はシミュレーション側のタスク削除機能の処理を示すフローチャートである。タスク削除要求を受け付けると、当該タスク削除要求に含まれるスレッドIDを獲得し、これを引数としてthr_delete()システムコールを行う(ステップS1102)。その後、当該スレッドに対応するTCBが使用していたメモリ領域を開放し、未使用状態とする。

【0054】[第3の実施形態] 第3の実施形態では、第1、2の実施形態におけるRTOSにおいて、更に割り込みを管理する機能を追加した場合を説明する。

【0055】(1)実機におけるRTOS
第1および第2の実施形態と同様に、モトローラ社製のMC68000を搭載した実機に対して本RTOSを動作させるものとし、以下では第3の実施形態で新たに追加された割り込みを管理する機能について説明する。

【0056】図15は、第3の実施形態による実機側のRTOSの構成を示すブロック図である。図9で示した構成に、割り込み管理機能120、割り込みコントローラ121、割り込みハンドラ122が加えられている。なお、割り込みコントローラとは、ハードウェア割り込みの制御を行うハードウェアである。AT互換機などでは8259AというLSIとして知られている。ハードウェア割り込みはこの割り込みコントローラを経由してCPUに伝えられ、発生した割り込みに対応した割り込みハンドラが起動される。図15における割り込みコントローラおよび割り込みハンドラの役割は、ハードウェア割り込みが生じたことを、当該RTOSの割り込み管理機能に伝えることである。

【0057】①割り込み管理機能120：割り込み開始要求を受けて、タスク中断機能109bを呼び出し、実行中のタスクのTCB114にコンテキストの保存を行う。割り込みが多重に起こった場合は、システムスタック123上にレジスタを積み上げていくことにより、多重割り込みに対応する。割り込み終了要求を受けて、スケジューリング機能106を呼び出した後、タスク再開機能115を呼び出して次タスクのTCB114からコンテキストの復帰を行う。多重割り込みからの復帰の場合は、システムスタック123上からコンテキストの復帰を行う。

【0058】図16は、実機側の割り込み処理を説明するフローチャートである。ハードウェア割り込みが発生すると、割り込みハンドラ122の先頭で割り込み管理機能120に対して割り込み開始要求を出す(ステップS802)。なお、割り込みハンドラ122の先頭とは、ハンドラルーチンが最初に処理するステップのことである。割り込み開始要求を受けた割り込み管理機能120では、まず、当該割り込み要求が多重割り込みか否かを判定する。多重割り

みであった場合は、ステップS805へ進み、システムスタック123にレジスタを待避させる。一方、多重割り込みでなければ、ステップS803からステップS804へ進み、タスク中断機能109を起動させる。なお、システムの属性値として、(多重)割り込み回数をカウントするカウンタが存在する。割り込み管理機能にて割り込み処理を開始する時にカウンタをインクリメントし、割り込み処理を終了する直前にカウンタをデクリメントする。多重割り込みか否かの判断はこのカウンタの値を元に行われる。また、ステップS805においてシステムスタック123に待避されるのは、処理中の割り込みに関わるレジスタ値である。例えば、現在処理中の割り込みAが、それより優先度の高い割り込みBにより中断された瞬間のレジスタ値をシステムスタックに保存し、高優先度の割り込みBの処理が終わった後に、割り込みAの処理を再開できるようにする。

【0059】次にステップS806において、割り込みハンドラ122の本体を実行させる。割り込み処理本体を実行した後、割り込みハンドラを終了する直前に、割り込み管理機能120に対して割り込み終了要求を出す(ステップS807)。なお、ステップS806では、割り込みが発生した要因に基づくI/O処理など、ハンドラの本来の処理が行われる。例えば、RS-232Cなどのシリアル通信では、シリアルポートヘデータがセットされた時に受信割り込みが発生し、対応する割り込みハンドラが起動されるが、割り込みハンドラでは受信したデータを読み込むためのI/O処理と読み取ったデータをバッファなどにつめる等の処理とが行なわれる。これが、ここで言うハンドラの本体の処理になります。S807はステップS802と同様に、割り込み管理機能呼び出すという動作そのものであり、具体的な処理としては、図16においてその下に描かれる破線の中の処理が行われることになる。

【0060】割り込み終了要求を受けた割り込み管理機能120は、ステップS808において、当該割り込みが多重割り込みか否かを判定する。多重割り込みであった場合は、ステップS811へ進み、システムスタック123からレジスタを復帰させて、ステップS812へ進む。一方、多重割り込みでない場合は、ステップS809へ進み、スケジューリング機能105を呼び出し、次に再開すべきタスクのタスクIDを獲得する。そして、ステップS810において、タスク再開機能110を起動する。そしてステップS812において、ハードウェア割り込み終了、すなわち割り込み復帰命令を実行して本処理を終える。割り込み復帰命令では、その割り込みレベルを解除し(一つ前のレベルに戻す)、システムスタック上に保存されているPC(プログラムカウンタ)や、SR(ステータスレジスタ)などを読み出して、割り込まれた箇所へ復帰を行なう。なお、その他のレジスタの復帰は事前に行なう必要がある。なお、ステップS80

6の割り込みハンドラ本体の処理の中では、本RTOSのシステムコールを実行する可能性がある。そのような場合、システムコールの実行によってシステムの状態が変化し、実行中だったタスクが資源待ちになったり、優先度の高いタスクが実行権を獲得するような状況が生じる。通常はシステムコールの終了の直前にスケジューリングされるが、割り込み処理中はタスクスイッチすることができないので、割り込み処理の終了の直前にスケジューリングを行う必要がある。従って、ステップS810で再開されるタスクは、割り込みの直前に実行されていたタスクとは限らない。

【0061】(2)シミュレーション環境におけるRTOS

第1の実施形態と同様、シミュレーション環境における実装例として、SunOSを搭載したワークステーション上で動作する本RTOSのシミュレータを動作させるもので、さらに追加した割り込み管理機能について説明する。

【0062】図17は、第3の実施形態によるシミュレーション装置側のRTOSの構成を示す図である。図17では、シミュレーション側のRTOSの基本的な構成に割り込み管理機能132を加えた構成が示されている。

【0063】①割り込み管理機能(スーパーシグナルスレッド)132:図17に示される割り込み管理機能132は、特定のシグナルを用いて、複数のハードウェア割り込みをシミュレートするしくみである。これは、アプリケーション・タスクとバインドするスレッド群とは独立して動く、割り込み管理専用のスーパーシグナルスレッドとして実現する。131は汎用OSからシグナルを受けた時に呼ばれるシグナルハンドラであり、シグナルハンドラ131はスーパーシグナルスレッド132に事象の発生を通知する。スーパーシグナルスレッド132は、多重割り込みをシミュレートするために独自のスタック133で割り込みレベルの管理をする。1つの割り込みハンドラは1つのスレッド(ハンドラスレッド)134として生成され起動される。つまり、多重割り込みが発生した場合は、多重のネスト回数と同数のハンドラスレッド134が生成されることになる。また、割り込みレベル制御およびハンドラスレッド134の管理はスーパーシグナルスレッド132が行い、多重割り込みのシミュレーションを行う。

【0064】図18は割り込みシグナル等の事象が発生した場合の割り込み管理機能132の処理の流れを示したフローチャートである。ステップS1601において、シグナルハンドラ131が割り込みの発生を検出すると、シグナルハンドラ131はスーパーシグナルスレッド132へ割り込みイベントの発生を通知する。

【0065】割り込みイベントの通知を受けたスーパーシグナルスレッド132は、当該割り込みが多重割り込みかどうかを判定する。多重割り込みであった場合は、ステップS

1602からステップS1604へ進み、実行中のハンドラスレッドを中断状態にしてスタック133にスタックする。一方、多重割り込みでなければ、ステップS1602からステップS1603へ進み、タスク中断機能109を起動させて、実行中のタスクを中断する。次に、ステップS1605において、対応するハンドラスレッドを生成する。

【0066】ステップS1606において、生成されたハンドラスレッド134が起動され、割り込みハンドラ本体が実行される。次にステップS1608において、多重割り込みかどうかを判定し、多重割り込みであったならばステップS1610へ進む。ステップS1610においては、中断中のハンドラスレッドをスタックから取り出し、実行状態にする。なお、中断中のハンドラ用スレッドを再開(実行状態にする)するのは、スーパーシグナルスレッド内のステップS1610で行われるが、実際にそのスレッドが再開するタイミングは汎用OSのスケジューリングに任される。通常、スーパーシグナルスレッドは最も優先度の高いスレッドとして生成するので、一連の割り込み処理はS1610、S1612まで流れ、終了する。その後、S1610で実行状態にされたスレッドが(汎用OSによって)再開される。

【0067】シミュレーション環境においては、割り込み管理機能もシミュレーションする必要があるため、多重割り込みのシミュレーションを行うには、割り込みハンドラをも中断/再開してコントロールしなければならない。そのため、汎用OSが提供するスレッドを生成して、そのスレッドでハンドラを実行し、中断/再開などの操作はスーパーシグナルスレッドが管理する。図中ステップS1606はハンドラ用のスレッドを生成するステップで、ステップS1607の割り込みハンドラ本体は、割り込みの要因に対する具体的な処理を記述した関数(もしくはブロック)である。

【0068】ステップS1608において多重割り込みでなかった場合(多重割り込みで最後の割り込み処理を得た場合も含む)、ステップS1609へ進み、スケジューリング機能105を読み出し、ステップS1611でタスク再開機能110を呼び出す。ここでシミュレーション側の割り込みはたとえば次のようにして発行させる。シミュレーション環境での割り込みをハンドリングするためには、独立して動くスレッドもしくはプロセスから、スーパーシグナルスレッドへ通知しなくてはならない。タイマ割り込みなどは、完全に独立して動くタイマ・スレッドを生成して、定期的にスーパーシグナルスレッドへ通知する。プロセス外からの割り込み要因の通知は、汎用OSが提供するシグナルハンドラを利用する。UNIXを例に説明すると、あらかじめRTOSのプロセスでシグナルハンドラを登録しておき、killコマンドなどで別のプロセスから当該プロセスにシグナルを送る。そして起動されたシグナルハンドラの中からスーパー

ーシグナルスレッドへ割り込みを通知する。

【0069】以上のように第3の実施形態によれば、RTOSを用いたアプリケーションのシミュレーションにおいて割り込み処理も扱えるようになる。

【0070】なお、上記各実施形態では、シミュレーション環境としてSunOSを用いた場合を説明したが、他のOS、例えば、Microsoft社のWindows 95、WindowNTなどのWin32-APIで提供されるマルチスレッド機能を用いて、同様なRTOSシミュレータを構築可能である。すなわち、上記実施形態で説明したRTOSは、第1乃至第3の実施形態で紹介したSunOSのシステムコール(thr_create(), thr_delete(), thr_suspend(), thr_resume()など)と同等なシステムコールを提供している汎用OS上で動作するシミュレータとして簡単に移植が可能である。

【0071】[第4の実施形態] 図19は、第4の実施形態によるRTOSの構造を示す図である。図19では、図1に示したRTOS(シミュレーション機側)にデバッグ情報を取得する機能が付加されている。

【0072】RTOSのシステムコールの実体は共通部分103にあり、各システムコールはすべて共通のシステムコール・エントリポイント1801を経由する構造になっている。そして、このシステムコール・エントリポイント1801は、トレース情報記録機能1802を有する。トレース情報記録機能1802は、システムコールが発行される毎に図20に示すような実行トレース情報901を生成する。すなわち、トレース情報記録機能1802は、発行されたシステムコールを取り出し、呼び出しタスクID902、システムコール実行情報903、タイムスタンプ904などの情報を取り出し、実行トレース情報901として一時的に保存する。ここで、呼び出しタスクID902は、タスクID又は、OS、割り込みの識別子を特定する情報である。システムコール実行情報903は、システムコール名(ID)およびパラメータ群を含む。更にタイムスタンプ904は、当該システムコール発行時のタイムカウンタの値を示す。

【0073】また、デバッグ情報転送機能1803は、保存した実行トレース情報901をグラフィカルに表示する可視化ツール(タスクスイッチ・ビューア)1806などの外部モジュールに転送する。

【0074】トレース情報記録機能1802は1回のシステムコールで図20に示す構造を持つ実行トレース情報を1つ記録するので、アプリケーションの一連の動作の履歴は大きな記憶容量を必要とする場合がある。加えて、一般的に実機環境ではメモリ領域などの資源が限られているため、データの記録方法になんらかの工夫が必要となる。このような問題に対処する方法として、例えば、リングバッファを用いた記録方法などが考えられる。リングバッファは、連続して生成される膨大なデー

タの中で最近のデータだけを効率良く記録する仕組みで、記録するデータ数がバッファサイズを超える場合に、最も古いデータから上書きしていく手法である。図19では、トレース情報記録機能1802で生成された実行トレース情報901をリングバッファ1807に保存していく例が示されている。リングバッファ1807はメモリ上に配置され、記録開始位置を指し示すStartポインタ1808と、記録終了位置を指し示すEndポインタ1809を用いてデータの記録、取り出し作業を行う。

【0075】図21は、トレース情報記録機能1802の処理の流れを示したフローチャートである。まず、システムコールの発行等により、ステップS1901において実行トレース情報が生成されると、ステップS1902において、リングバッファ1807のEndポインタ1809が示す位置に当該実行トレース情報を書き込む。次にステップS1903において、有効データ数がバッファサイズ以上になっているかを判定する。有効データ数がバッファサイズより小さければステップS1905へ進み、有効データ数を1つインクリメントする。一方、有効データ数がバッファサイズ以上となっている場合は、Startポインタ1808をインクリメントする。このときStartポインタ1808の位置がオーバーランした場合は先頭に戻る。そして、ステップS1906において、Endポインタ1809の位置をインクリメントし、オーバーランした場合は先頭に戻る。なお、リングバッファは通常の物理的に線形なバッファを用いて、論理的にリング状に見えるような仕組みを持つものである。Start位置やEnd位置を指し示すポインタは、操作の度にインクリメントされるが、いずれ物理バッファの最後尾に到達する。さらにインクリメントするとバッファをオーバーランしてしまうので、その時点でポインタをバッファの先頭にリセットする。この仕組みにより、論理的にリング状に見せかけることができる。

【0076】また、デバッグ情報転送機能1803は、図1における共通部分103の内部に実装されるが、物理的な通信経路はターゲットシステムに依存する。そのため、ターゲットシステムが備えている通信手段を利用するために、ハードウェア依存部分106にデバッグ通信モジュール1805を具備する。デバッグ通信モジュール1805は、バッファのread/writeを行うI/Fを提供する。最下層のレイヤとしては具体的には、シリアル、ソケット、共有メモリ、ICE、ROMエミュレータなどが考えられる。

【0077】図22はデバッグ情報転送機能1803において、先の例で示したリングバッファを用いた場合の処理の手順を示したフローチャートである。まず、デバッグ情報転送機能1803は、ステップS2002において有効データ数が1以上であるか否かを判断する。有効データ数が1以上であれば、転送すべき実行トレース

情報が存在すると判断し、ステップS2003へ進む。ステップS2003では、リングバッファからデータを1つ取り出す。そして、ステップS2004において、取り出した実行トレース情報をデバッグ通信モジュール1805を介して外部モジュール（例えば可視化ツール1806）へ転送する。そして、ステップS2005において有効データ数を1つ減少させて、ステップS2002へ戻る。

【0078】この様にして転送したトレース情報をグラフィカルに表示する表示装置として、図23にタスクスイッチ・ビューア1806の概観の一例を示す。

【0079】この様に、RTOSのデバッグ情報として有効なシステムコールやタスクスイッチの履歴、システム管理情報などを取得する手段を共通部分で実装することにより、異なるプラットフォーム間でも統一された思想でデバッグを行うことができる。

【0080】以上のように、第4の実施形態のRTOSは、ハードウェアに依存する部分を必要最小限となるように切り出し、それ以外の部分はさまざまなプラットフォームで共通に利用できるようにC言語で書かれており、ハードウェアの違いによる影響が少なくなるような構造を持つものであり、システムコールはすべて共通のエントリポイントを経由してコールされ、システムコール実行後にスケジューリング機能が呼び出される構造を持つ。そして、この共通エントリポイントにおいて

(1) 呼びだしタスクIDと(2)システムコール実行情報と(3)タイムスタンプなどの情報を保存するトレース情報記録機能と、保存したトレース情報をタスクスイッチ・ビューアなどの外部モジュールに転送する手段とを設けたことにより、プラットフォームに依存しないデバッグ情報を提供することが可能である。

【0081】以上説明したように、上記各実施形態によれば、次のような効果が期待できる。すなわち、

(1) プラットフォーム非依存性
アプリケーション・タスクをまったく修正せずに、実機と汎用開発マシンなど異なるプラットフォーム上で同一の振る舞いをさせることができる。

(2) 移植の容易性
機種依存部分が最小限に切り出されているので、他のプラットフォームへの移植が容易に実現できる。

(3) プラットフォーム非依存なデバッグ
ハードウェアに依存する物理的通信を行うモジュールだけ実装すれば、様々なプラットフォームで、同一のデバッグが利用できる。

(4) デバッグ情報の統一化
取得するデバッグ情報はプラットフォームに依存しない情報なので、異なるプラットフォーム間でも統一思想でデバッグを行うことができる。

【0082】なお、本発明の目的は、前述した実施形態の機能を実現するソフトウェアのプログラムコードを記

録した記憶媒体を、システムあるいは装置に供給し、そのシステムあるいは装置のコンピュータ（またはCPUやMPU）が記憶媒体に格納されたプログラムコードを読み出し実行することによっても、達成されることは言うまでもない。

【0083】この場合、記憶媒体から読出されたプログラムコード自体が前述した実施形態の機能を実現することになり、そのプログラムコードを記憶した記憶媒体は本発明を構成することになる。

【0084】プログラムコードを供給するための記憶媒体としては、例えば、フロッピディスク、ハードディスク、光ディスク、光磁気ディスク、CD-ROM、CD-R、DVD、磁気テープ、不揮発性のメモ리카ード、ROMなどを用いることができる。

【0085】また、コンピュータが読出したプログラムコードを実行することにより、前述した実施形態の機能が実現されるだけでなく、そのプログラムコードの指示に基づき、コンピュータ上で稼働しているOS（オペレーティングシステム）などが実際の処理の一部または全部を行い、その処理によって前述した実施形態の機能が実現される場合も含まれることは言うまでもない。

【0086】さらに、記憶媒体から読出されたプログラムコードが、コンピュータに挿入された機能拡張ボードやコンピュータに接続された機能拡張ユニットに備わるメモリに書込まれた後、そのプログラムコードの指示に基づき、その機能拡張ボードや機能拡張ユニットに備わるCPUなどが実際の処理の一部または全部を行い、その処理によって前述した実施形態の機能が実現される場合も含まれることは言うまでもない。

【0087】

【発明の効果】以上説明したように、本発明のマルチタスク制御方法によれば、あらゆるプラットフォーム上で同一のインターフェース仕様を持ち、アプリケーションが同一の動作を行うことが可能となる。

【0088】また、本発明によれば、あらゆるプラットフォーム上で同一のインターフェース仕様を持ち、アプリケーションが同一の動作を行うことを可能とするRTOS上のデバッグ環境（タスクスイッチ・ビューア等）において、あらゆるプラットフォーム上で統一的なデバッグ情報を収集することが可能となる。

【0089】

【図面の簡単な説明】

【図1】第1の実施形態におけるRTOSの基本的な構成図である。

【図2】システムコール発行からタスクスイッチまでの、共通部分による処理の手順を表したフローチャートである。

【図3】プライオリティ順のスケジューリングアルゴリズムを実現した一例を説明する図である。

【図4】タイムシェアリングによるスケジューリングを

実現した一例を示す図である。

【図5】実機側におけるタスク中断機能の処理の流れを表すフローチャートである。

【図6】実機側におけるタスク再開機能109の処理の流れを表すフローチャートである。

【図7】シミュレーション機側におけるタスク中断機能の処理の流れを表すフローチャートである。

【図8】シミュレーション機側のタスク再開機能の処理の流れを表すフローチャートである。

【図9】第2の実施形態による実機側のRTOSの構成を示すブロック図である。

【図10】実機側のタスク生成機能の処理を説明するフローチャートである。

【図11】実機側のタスク削除機能の処理を表すフローチャートである。

【図12】第2の実施形態によるシミュレーション側のRTOSの構成を示すブロック図である。

【図13】シミュレーション側のタスク生成機能の処理手順を表すフローチャートである。

【図14】シミュレーション側のタスク削除機能の処理を示すフローチャートである。

【図15】第3の実施形態による実機側のRTOSの構成を示すブロック図である。

【図16】実機側の割り込み処理を説明するフローチャートである。

【図17】第3の実施形態によるシミュレーション装置側のRTOSの構成を示す図である。

*

*【図18】シグナル等の事象が発生した場合の割り込み管理機能132の処理の流れを示したフローチャートである。

【図19】第4の実施形態によるRTOSの構造を示す図である。

【図20】実行トレース情報のデータ構成例を示す図である。

【図21】トレース情報記録機能1802の処理の流れを示したフローチャートである。

【図22】デバッグ情報転送機能1803において、先の例で示したリングバッファを用いた場合の処理の手順を示したフローチャートである。

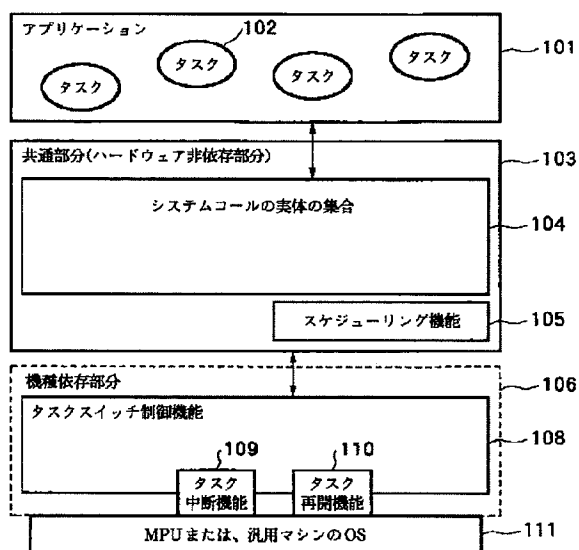
【図23】タスクスイッチ・ビューア1806の概観の一例を示す図である。

【図24】第1の実施形態における実機とシミュレーション環境を提供するコンピュータ装置を示す図である。

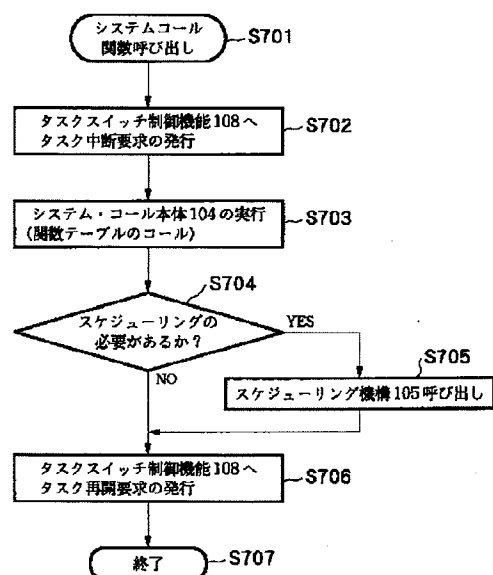
【符号の説明】

- 101 アプリケーション
- 102 アプリケーション・タスク
- 103 RTOSの共通部分
- 104 RTOSシステムコールの実体の集合
- 105 スケジューリング機能
- 106 RTOSの機種依存部分
- 108 タスクスイッチ制御機能
- 109 タスク中断機能
- 110 タスク再開機能

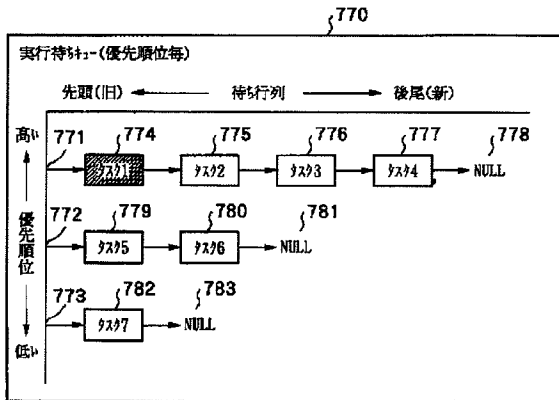
【図1】



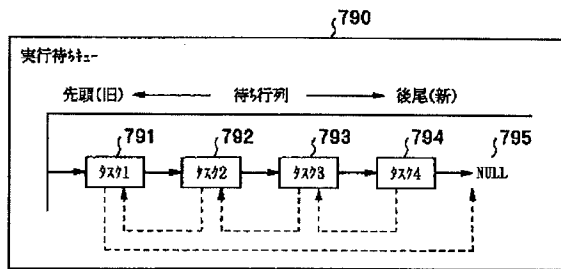
【図2】



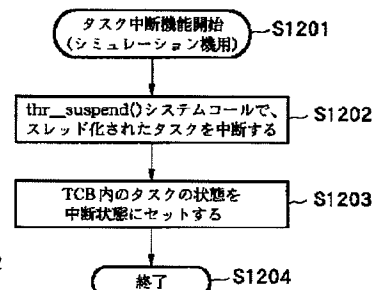
【図3】



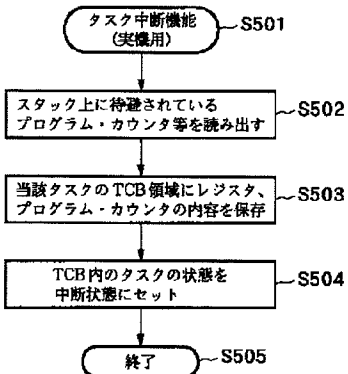
【図4】



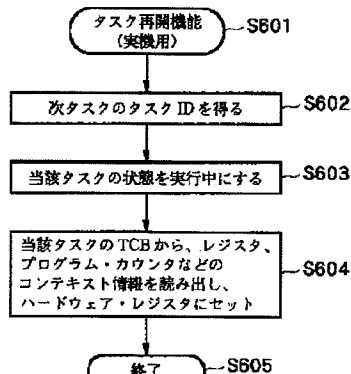
【図7】



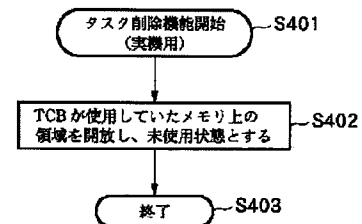
【図5】



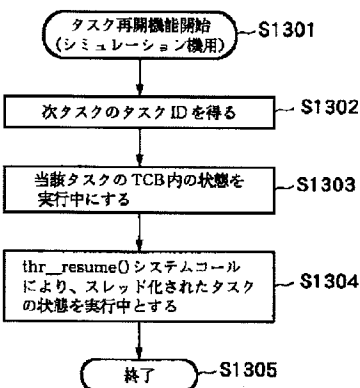
【図6】



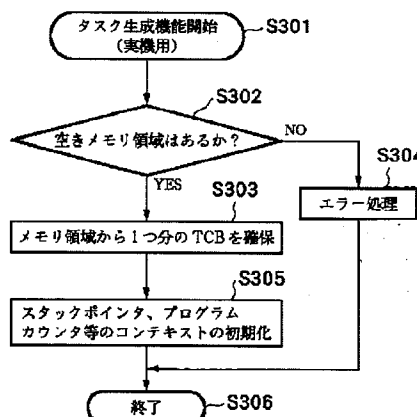
【図11】



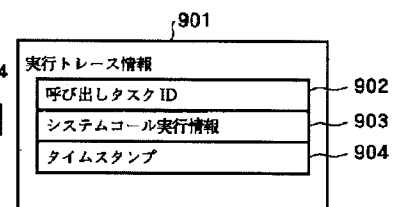
【図8】



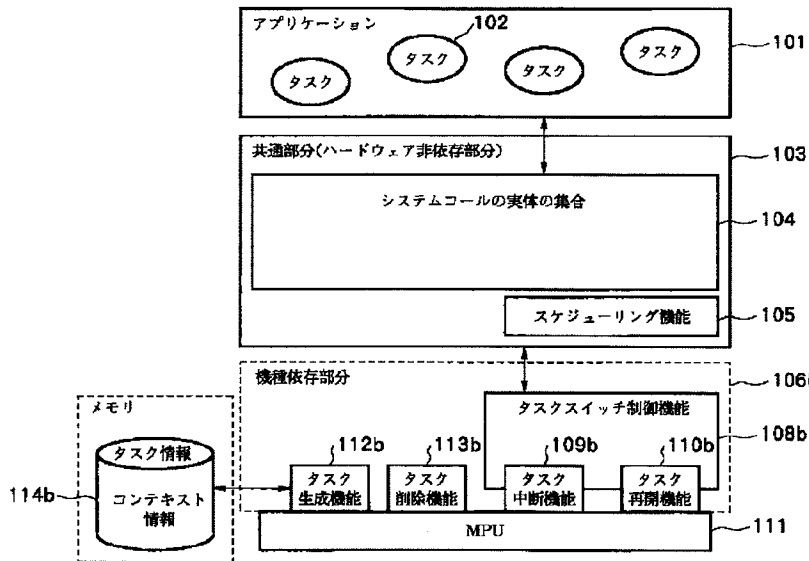
【図10】



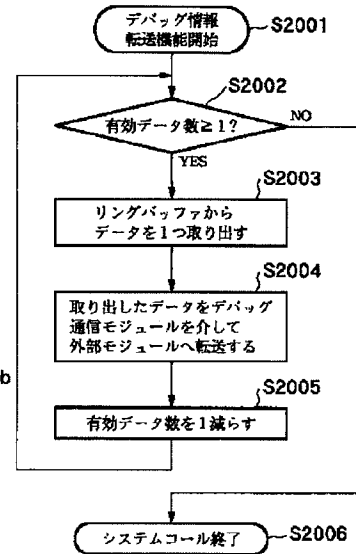
【図20】



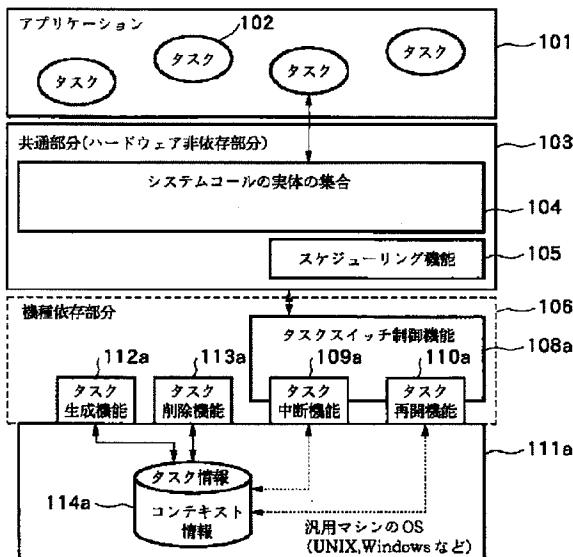
【図9】



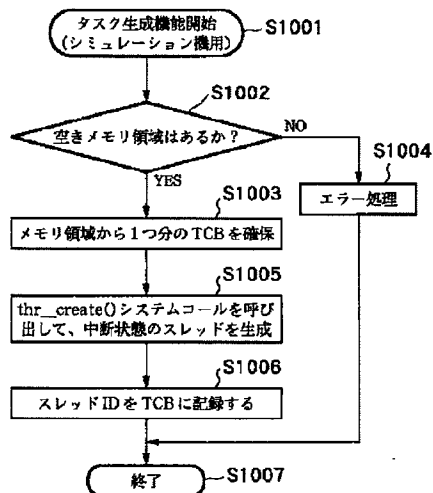
【図22】



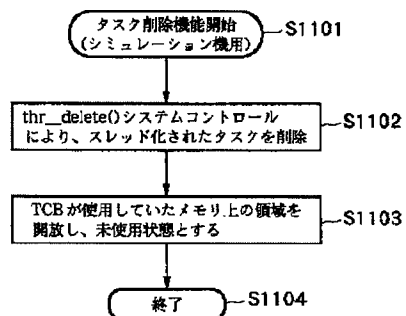
【図12】



【図13】



【図14】



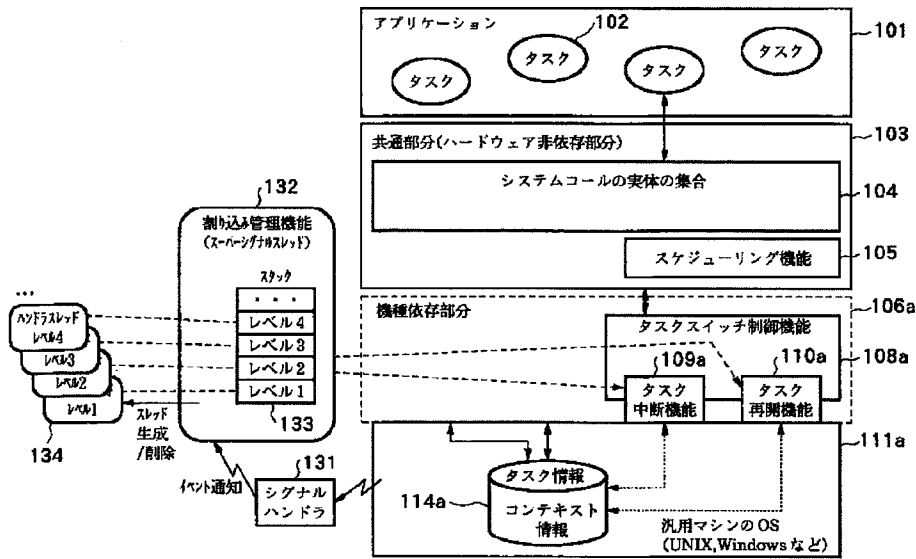
```

graph TD
    S801([ハードウェア  
割り込みの発生]) --> S802[割り込み管理機能 120へ  
割り込み開始要求の発行]
    S802 --> S803{多重割り込みか?}
    S803 -- YES --> S805[システムスタック 123に  
レジスタを待避]
    S803 -- NO --> S804[タスク中断機能 109の呼び出し]
    S805 --> S804
    S804 --> S806[割り込みハンドラ  
本体 122の実行]
    S806 --> S807[割り込み管理機能 120へ  
割り込み終了の発行]
    S807 --> S808{多重割り込みか?}
    S808 -- YES --> S811[システムスタック 123から  
レジスタを復帰]
    S808 -- NO --> S809[スケジューリング機構  
105呼び出し]
    S809 --> S810[タスク再開機能 110の呼び出し]
    S811 --> S810
    S810 --> S812([ハードウェア  
割り込み終了])
  
```

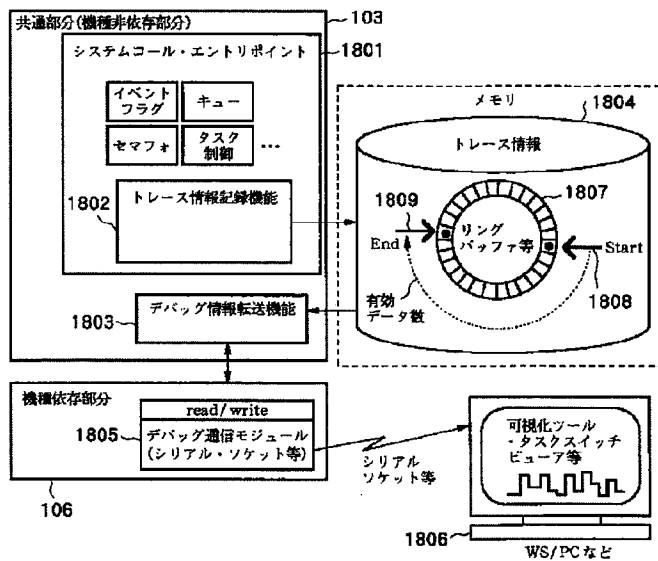
```

graph TD
    S1600([シグナルの発生等]) --> S1601[シグナルハンドラ  
スーパーシグナルスレッド  
132へ事象発生通知]
    S1601 --> S1602{多重割り込みか?}
    S1602 -- YES --> S1604[実行中のハンドラ  
スレッドを中断状態に  
してスタック133につむ]
    S1602 -- NO --> S1603[タスク中断機能109の呼び出し]
    S1604 --> S1603
    S1603 --> S1605[対応するハンドラスレッド  
を生成]
    S1605 --> S1606[ハンドラスレッドが起動]
    S1606 --> S1607[割り込みハンドラ本体の実行]
    S1607 --> S1608{多重割り込みか?}
    S1608 -- YES --> S1610[実行中のハンドラ  
スレッドを中断状態に  
してスタック133につむ]
    S1608 -- NO --> S1609[スケジューリング機構  
105呼び出し]
    S1609 --> S1611[タスク再開機能110の呼び出し]
    S1610 --> S1611
    S1611 --> S1612([ハードウェア  
割り込み終了])
  
```

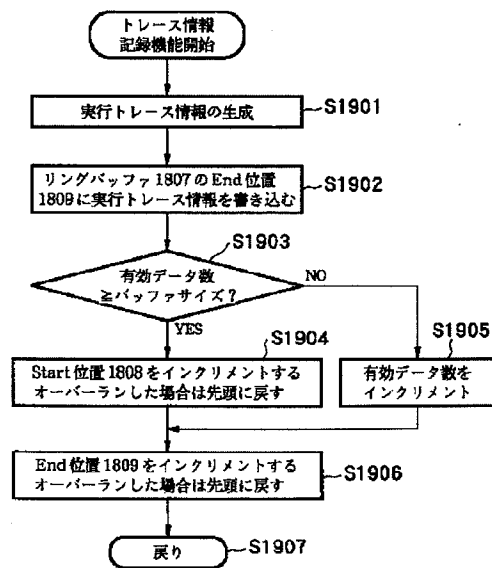

【図17】



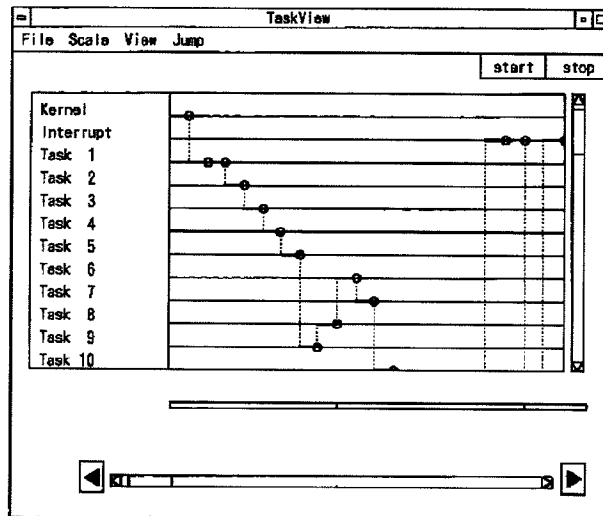
【図19】



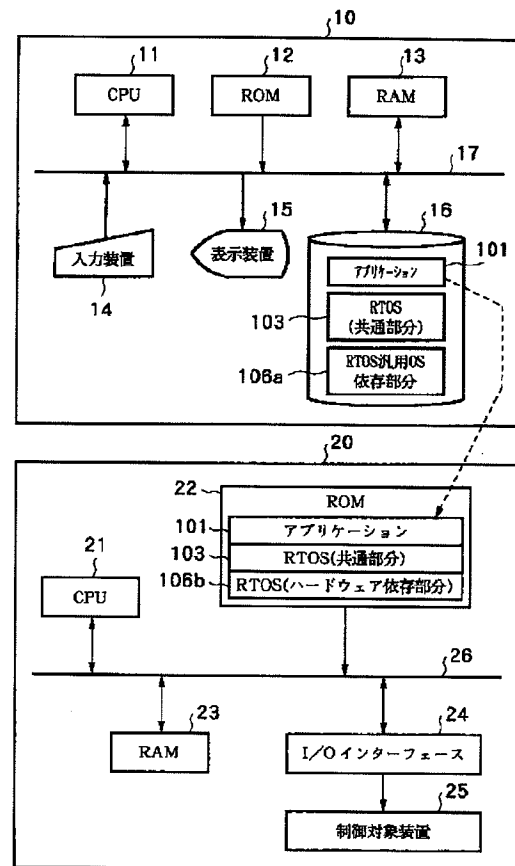
【図21】



【図23】



【図24】



フロントページの続き

(72)発明者 岩瀬 洋一
東京都大田区下丸子3丁目30番2号 キヤ
ノン株式会社内

(72)発明者 山田 潤二
東京都大田区下丸子3丁目30番2号 キヤ
ノン株式会社内
Fターム(参考) 5B098 GA02 GA04 GA08 GB02 GB09
GB11 GB13 GC03 GC14 JJ07